1980

Introduction to the Configurable, Highly Parallel (CHiP) Computer)

Lawrence Snyder

Report Number:
80-351

Introduction to the Configurable, Highly Parallel Computer

Lawrence Snyder
Department of Computer Sciences
Purdue University
West Lafayette, IN
47907

Abstract: The Configurable, Highly Parallel (CHiP) Computer family is introduced. These architectures are built around a lattice of programmable switches and data paths that permit processing elements to be connected in arbitrary patterns. The approach preserves locality. The parameters that determine various family members are discussed including switch configuration storage capacity, switch and processor element degrees and corridor width. An efficient embedding of a complete binary tree is presented to illustrate interconnection pattern programming. An algorithm for solving a system of linear equations is given to illustrate the versatility of configurability.

CSD-TR-351
November 1980
Revised May 1981

The research described herein is part of the Blue CHiP Project. Funding is provided in part by the Office of Naval Research under Contract N00014-80-K-0816 and Contract N00014-81-K-0360, Special Research Opportunities Program Task SRO-100.

Introduction

> *polymorphism*, n.(1): capability of assuming different forms; capability of wide variation.
>
> -Webster's Third International Dictionary-

When von Neumann computers were still new and exciting, scientists noted in popular accounts that unlike mechanical machines, computers are polymorphic - their function can be radically changed simply by changing programs. Polymorphism *is* fundamental, but it quickly became familiar to the point of being obvious and has been mentioned little since, even though it has continued to underlie important advances such as time-sharing and programmable microcode. Now, as we are confronted with the potential for highly parallel computers made possible by very large scale integrated (VLSI) circuit technology, we may ask:

What is the role of polymorphism in parallel computation?

To answer this question, we must review the characteristics of parallel processing and the benefits and limitations of VLSI technology.

Algorithmically Specialized Processors

Perhaps the most important property of VLSI circuit technology is that the manufacturing processes use photolithographic means to create copies of a circuit. Fabrication by photolithography (or the newer X-ray lithography techniques) requires a fixed number of steps to produce a circuit, independent of the circuit's complexity. It costs no more to make copies of a chip containing a NAND gate than to make copies of a chip containing a microprocessor, although yields will likely be higher for the former and wire bonding costs higher for the latter. Preparing and debugging the lithographic masks is expensive, so the technology favors parallel processing techniques that employ many copies of the same, possibly complex circuit.

Recognition of uniformity as the source of leverage in VLSI caused a flurry of research during the past half decade. This research resulted in a number of device proposals which we may call *algorithmically specialized processors*. By focusing on computationally intensive problems and carefully dissecting algorithms for them, researchers have developed algorithmically specialized processors having several important characteristics:

- construction is based on a few easily tessellated processing elements,
- locality is exploited, that is, data movement is often limited to adjacent processing elements,
- pipelining is used to achieve high processor utilization.

Examples of algorithmically specialized processors include designs for LU decomposition [2,5] (the main step in solving systems of linear equations), the solution of linear recurrences [2], tree processors [4,5,6] (used in

searching, sorting and expression evaluation), dynamic programming [7] (a general problem solving technique with numerous applications), join processing [8] (for data base querying), and many others.

Algorithmically specialized processing components must be joined together to solve a large, computationally intensive problem. This *composition* step is crucial since whole problems tend to be multiphased and these components tend to be specialized to an algorithm used in only one phase. For example, to solve a system of linear equations ($Ax=b$) one might use a processor component to form the LU decomposition of the matrix A ($A=LU$) and then use a linear recurrence solver component to perform the substitution phases ($Ly=b$ and $Ux=y$). As another example, queries in data base query languages are formed by composing operations such as "search" and "join".

If the component processors are implemented on chips, one way to compose them is to wire them together. This solution is inflexible since the components are dedicated to a particular problem and cannot be used for another problem. Another compositional scheme is to join the processors to a bus as "pheripherals." This is more flexible since a processor can be used in different phases, but the bus becomes a bottleneck and time is wasted in interphase data movement.

A more flexible approach is to replace the dedicated processing elements with more general microprocessors and simply to program the algorithmically specialized processing function. This solution is much more flexible since different components can use the same devices by changing programs (provided the interconnection pattern is the same). The bus bottleneck is eliminated. There is a loss in performance with this

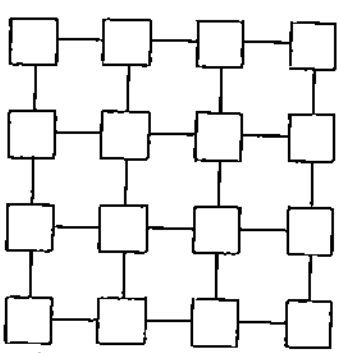
polymorphism, since circuit implementation of the primitive actions is replaced by the slower process of instruction execution.

But the main problem with this approach is that algorithmically specialized processors often use different interconnection structures (see Figure 1). There is no guarantee that the consecutive phases of the computation can be done efficiently in place. For example, if we have an $n \times n$ mesh connected microprocessor structure and want to find the maximum of n^2 elements stored one per processor, $2n-1$ steps are necessary and sufficient to solve the problem. But a faster algorithmically specialized processor for this problem uses a tree interconnection pattern to find the solution in $2 \log n$ steps. For large n this is a benefit worth seeking. Again, a bus can be introduced to link several differently connected multiprocessors including mesh and tree connected multiprocessors. Data could be transferred when a change in the processor structure would be beneficial. But the bottleneck is quite serious - in the example, data has to be transferred at a rate proportional to $n^2/\log n$ words per step to make the transfer worthwhile. What we need is a multiprocessor with more polymorphism that does not compromise the benefits of VLSI technology.

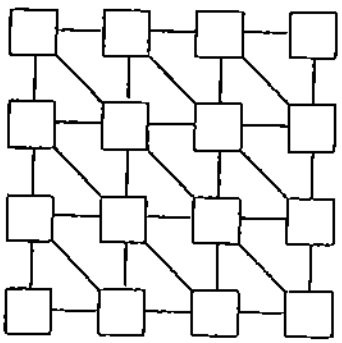
The Configurable, Highly Parallel (CHiP) computer is a multiprocessor architecture that provides a programmable interconnection structure integrated with the processing elements. Its objective is to provide the flexibility needed to compose general problem solutions while retaining the benefits of uniformity and locality that the algorithmically specialized processors exploit.

The CHiP Architecture Overview

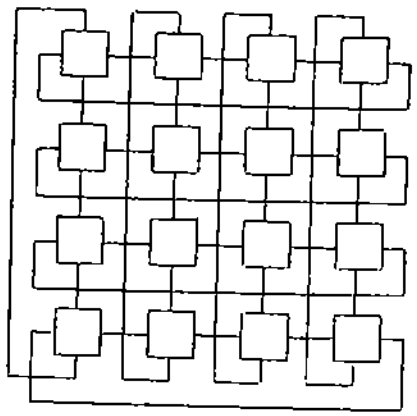
The CHiP computer is a family of architectures each constructed from



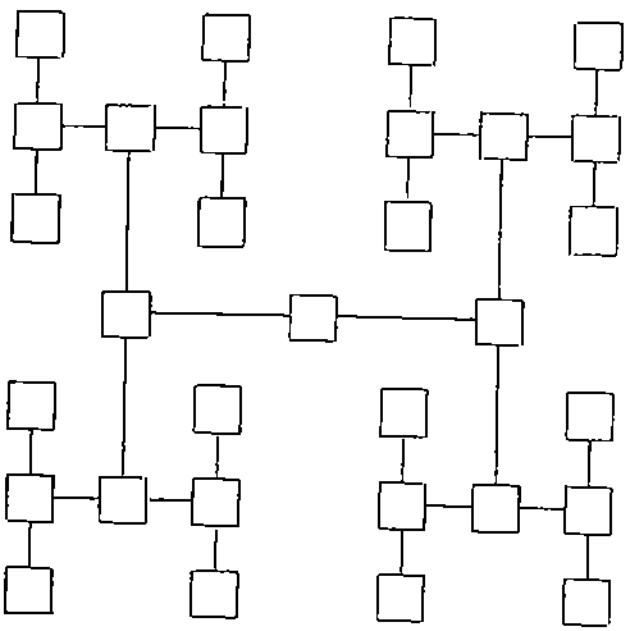
(a)



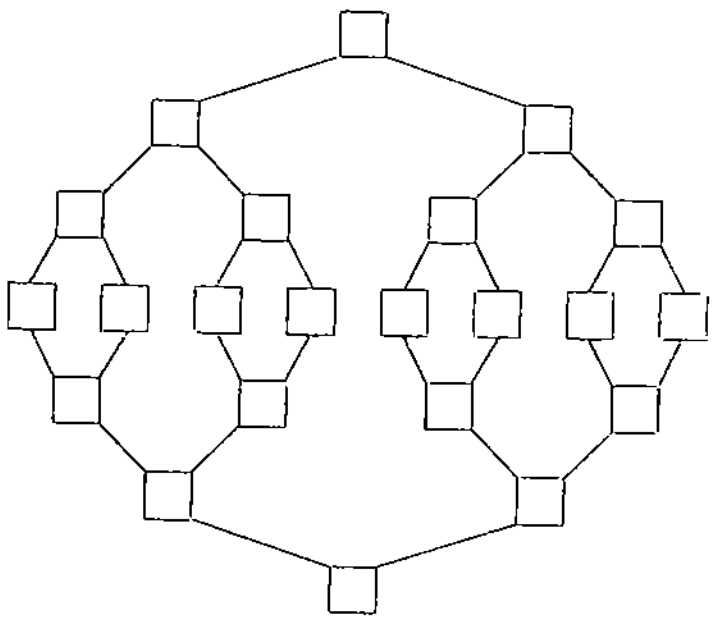
(b)



(c)



(d)



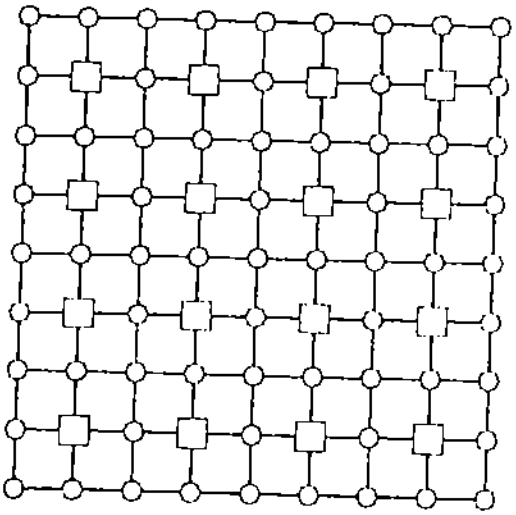
(e)

Figure 1. Interconnection patterns for algorithmically specialized processors: (a) mesh, used for dynamic programming [7]; (b) hexagonally connected mesh used for LD decomposition [2]; (c) torus used for transitive closure [7]; (d) binary tree used for sorting [4]; (e) double tree used for searching [5].

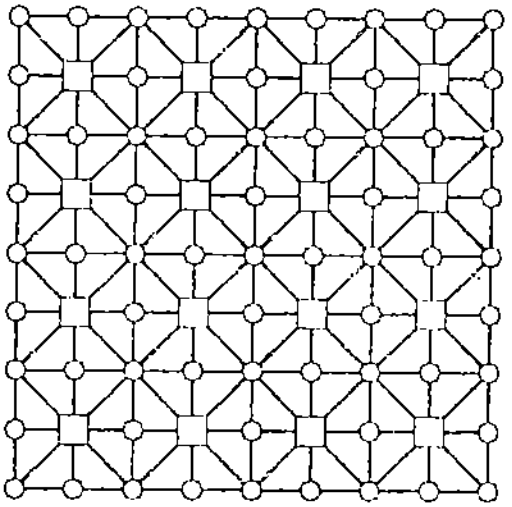
three components: (a) a collection of homogeneous microprocessors, (b) a switch lattice and (c) a controller. The switch lattice is the most important component and the main source of differences among family members.

The *switch lattice* is a regular structure formed from programmable switches connected by data paths. The microprocessors (hereafter called processing elements or PEs) are not directly connected to each other, but rather are connected at regular intervals to the switch lattice. Figure 2 shows three examples of switch lattices. Generally, the layout will be square although other geometries are possible. The perimeter switches are connected to external storage devices. A production CHiP computer might have from 2^8 to 2^{16} PEs. (With current technology only a few PEs and switches can be placed on a single chip. As improvements in fabrication technology permit higher device densities per unit area, a single chip can host a larger region of the switch lattice. Moreover, as discussed below, the CHiP architecture is quite suitable for "wafer level" fabrication.)

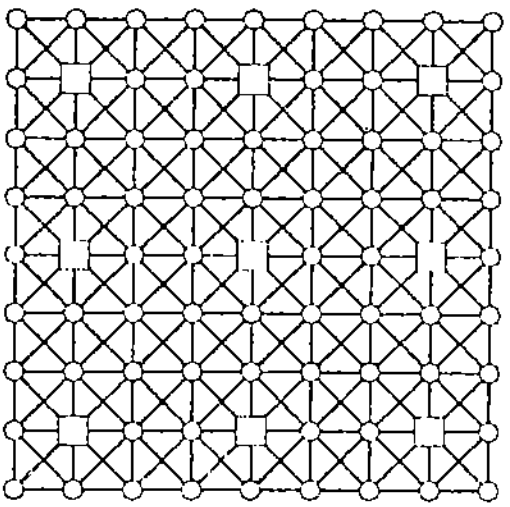
Each switch in the lattice contains local memory capable of storing several configuration settings. A configuration setting enables the switch to establish a direct, static connection among two or more of its incident data paths. (Notice, this is circuit switching rather than packet switching.) For example, we achieve a mesh interconnection pattern of the PEs for the lattice in Figure 2(a) by assigning North-South configuration settings to alternate switches in odd numbered rows and East-West settings to switches in the odd numbered columns. Figure 3 illustrates the configuration; Figure 4 gives the configuration settings of a binary tree.



(a)



(b)



(c)

Figure 2. Three switch lattice structures. Circles represent switches; squares represent PE's.

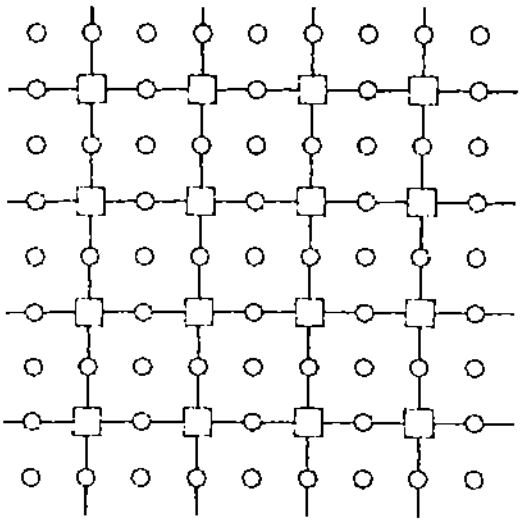


Figure 3. The switch lattice of Figure 2(a) configured into a mesh pattern.

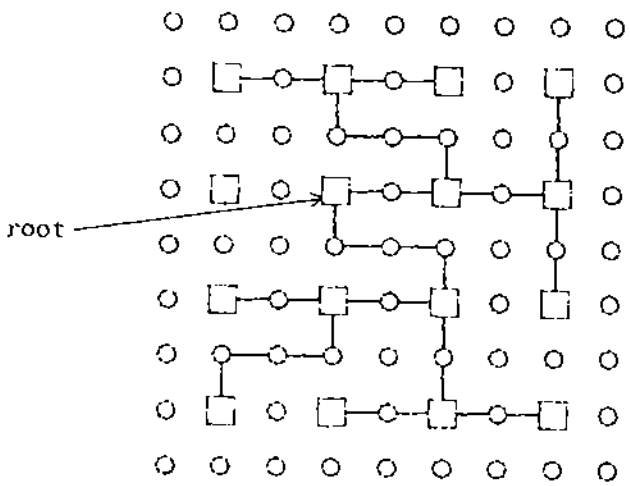


Figure 4. The switch lattice of Figure 2(a) configured into a binary tree.

The controller is responsible for loading the switch memory. (This task is performed via a separate interconnection "skeleton" that is transparent to this discussion.) The switch memory is loaded preparatory to processing and is performed in parallel with the PE program memory loading. Typically, program and switch settings for several phases can be loaded together. The chief requirement is that the local configuration settings for each phase's interconnection pattern be assigned to the same memory location in all switches. For example, in each switch, location 1 might be used to store the local configuration to implement a mesh pattern, location 2 might store the local configuration for the tree interconnection pattern, etc.

CHiP processing begins with the controller broadcasting a command to all switches to invoke a particular configuration setting. For example, suppose it is the setting stored at location 1 that implements a mesh pattern. With the entire structure interconnected into a mesh, the individual PEs synchronously execute the instructions stored in their local memory. PEs need not know to whom they are connected; they simply execute instructions such as READ EAST, WRITE NORTHWEST, etc. The configuration remains static. When a new phase of processing is to begin, the controller broadcasts a command to all switches to invoke a new configuration setting, say the one stored at location 2 implementing a tree. With the lattice restructured into a tree interconnection pattern, the PEs resume processing, having spent only a single logical step in interphase structure reconfiguration.

The overview of the CHiP computer family has been superficial, but it has provided a context in which to present a more thorough treatment.

The next three sections are:

A closer look, giving details about switches, lattices and the controller

Embedding an interconnection structure, an example of how to configure the lattice into a complete binary tree, and

Solving a system of linear equations, illustrating how a multiphased problem might be solved.

We conclude with a *Discussion* section in which we mention some of the consequences of the CHiP architecture approach.

A Closer Look

We consider some of the characteristics that distinguish members of the family of CHiP computers.

Switches. It is convenient to think of switches as being defined by several parameters.

- m - the number of wires entering a switch on one data path, or data path width,
- d - the degree, or number of incident data paths,
- c - the number of configuration settings that can be stored in a switch.

The value of m reflects the balance struck between parallel and serial data transmission. This balance will be influenced by several considerations, one of which is the limited number of pins on the package containing the chips of the CHiP lattice. Specifically, if a chip hosts a square region of the lattice containing n PEs, then the number of pins required is proportional to $m\sqrt{n}$.

The value of d will usually be 4, as in Figure 2(a), or 8, as in Figure 2(c). Figure 2(b) shows a mixed strategy which exploits the fact that switches tend to be used in two different roles. Switches at the intersection of the vertical and horizontal switch corridors tend

to perform most of the routing while those interposed between two adjacent PEs act more like extended PE ports for selecting data paths from the "corridor buses". Specializing the degree of the switch to these activities reduces the number of bits required to specify a configuration setting and thus saves area.

The value of c is influenced by the number of configurations that are likely to be needed for a multiphase computation and the number of bits required per setting. This latter number depends on the degree and the crossover capability of the switch.

"Crossover capability" is a property of switches referring to the number of distinct data path groups that a switch can simultaneously connect. We speak of data path "groups" rather than data path pairs since fanout is permitted at a switch, i.e. a switch can connect more than a pair of data paths. Crossover capability is specified by an integer g in the range 1 to $d/2$. Thus 1 indicates no crossover and $d/2$ is the maximum number of distinct paths intersecting at a degree d switch. Like the three parameters mentioned above, the crossover capability g is fixed at fabrication time.

The number of bits of storage needed for a switch is modest, dgc. This provides a bit for each direction for each crossover group for each configuration setting. A technique to reduce this value is to provide for the loading of switch settings while the CHiP processor is executing. This quality, called "asyncronous loading", permits a smaller value of c by taking advantage of two facts: algorithms often use configurations that differ in only a few places, and configurations often remain in effect long enough to provide time to prepare for future settings.

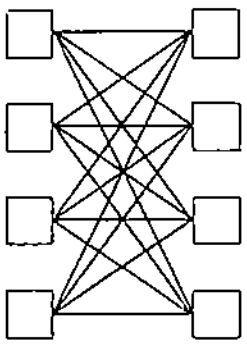
Lattice. From Figure 2 it is clear that lattices can differ in several characteristics. The PE degree, like the switch degree, is the

number of incident data paths. Most algorithms of interest use PEs of degree eight or less. Larger degrees are probably not necessary since they can be achieved either by multiplexing data paths or, with some loss in PE utilization, by logically coupling processing elements, e.g. two degree four PEs could be coupled to form a degree six PE where one serves only as a buffer.

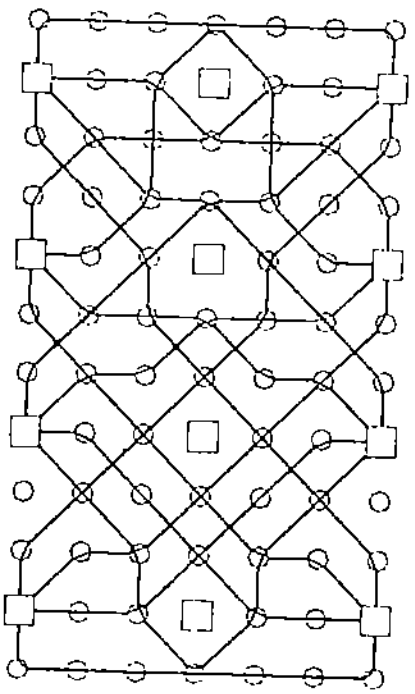
Call the number of data paths that separate two adjacent PEs the *corridor width*, w. (See Figure 2(c) for a $w = 2$ lattice.) This is perhaps the most significant parameter of a lattice since it influences the efficiency of PE utilization, the convenience of interconnection pattern embeddings, and the overhead required for the polymorphism.

To see the impact of corridor width, let us embrace graph embedding parlance and say that a switch lattice *hosts* a PE interconnection pattern. In theory, even the simplest lattice (like the one in Figure 2(a)) can host an arbitrary interconnection pattern. But to do so may require the PEs to be underutilized for two reasons. First PEs may be coupled to achieve high PE degree as mentioned at the beginning of this section. Second, and more importantly, adjacent PEs in the (logical) guest interconnection pattern may have to be assigned to widely spaced PEs in the hosting lattice (i.e. separated by unused PEs) in order to provide sufficiently many data paths for the edges. (Figure 5 shows the embedding of the complete bipartite graph, $K_{4,4}$, in the lattice of Figure 2(c) where the center column of PEs is unused.) Increasing corridor width improves processor utilization when complex interconnection patterns must be embedded since it provides more data paths per unit area.

How wide should corridors be? It all depends on which interconnection patterns are likely to be hosted and how economically necessary it is to maximize PE utilization. For most of the algorithmically specialized



(a)



(b)

Figure 5. Graph $K_{4,4}$ shown in (a) is embedded into the lattice of Figure 2(c) using a switch with crossover value $g = 2$.

processors developed for VLSI implementation, a corridor width of two suffices to achieve optimal or near optimal PE utilization. However, to be sure of hosting all planar interconnection patterns of n nodes with reasonably complete processor utilization, a width proportional to $log\ n$ suffices and may be necessary [9]. To host patterns such as the shuffle-exchange graph with high efficiency will require still wider corridors, on the average w must be at least proportional to $n/log\ n$ [10].

Selecting a corridor width is a difficult decision, especially if it is a nonconstant width. The benefit is higher PE utilization in some cases; the cost is a loss of some locality in all cases, introduction of more area overhead, and increased problems with "pin" limitations. Preliminary evidence indicates that $w \leq 4$ provides a reasonable cost/benefit tradeoff, but further experimentation and analysis are required. (See reference [12] for an elaboration of this discussion.)

Embedding an Interconnection Pattern

In addition to the conventional polymorphism derived from PE programming, we have provided for a second kind of polymorphism - the programmable switches. This requires us to provide for interconnection pattern programming, i.e. the specification of a global interconnection pattern. When viewed in a programming language context, the "source program" is a global interconnection pattern that a compiler translates into an "object code" of individual switch settings suitable for loading into the switches by the CHiP controller. The general programming language and compiler issues need not concern us here, however, for we will explore only one particular interconnection pattern: the complete binary tree. This example will enable us to illustrate the differences between

embedding into the plane and embedding into the CHiP lattice.

The complete binary tree has 2^p-1 PE's, one at each node. One possible layout of this structure in the CHiP lattice is a direct translation of the "hyper-H" strategy [1] illustrated in Figure 1(d). Figure 6 illustrates this embedding into the lattice of Figure 2(a) and it is clear that a significant number (approaching one half) of the PEs are unused in this naive approach. The problem is that although the hyper-H is an excellent embedding on plain silicon where the placement of PEs and data paths is arbitrary, CHiP lattice embeddings must conform to the prespecified PE and data path sites. As we shall see, this constraint is not onerous.

To illustrate an optimal embedding (in terms of maximizing the use of PEs), assume that we have an $n \times n$ CHiP lattice where $n = 2^k$ for some integer k. This gives 2^{2k} PEs, so a binary tree of depth $2k$ fits with only one unused PE, since it has $2^{2k}-1$ nodes. Call this unused PE a "spare."

We proceed inductively by pairing two embedded subtrees to form a new tree one level higher. For the basis of the induction it is convenient to use a three node binary tree embedded with one spare in a 2×2 portion of the lattice. Pairing square subtree embeddings produces rectangles with sides in ratio 2:1. Pairing these rectangles yields squares again. In general we pair two subtrees each with $2^{2k}-1$ nodes and a spare to produce a new $2^{2k+1}-1$ node tree in which one of the subtree spares becomes the root of the new tree and the other spare becomes the spare of the new tree. The interesting problem is to place the spares at the proper sites for the next step in the induction.

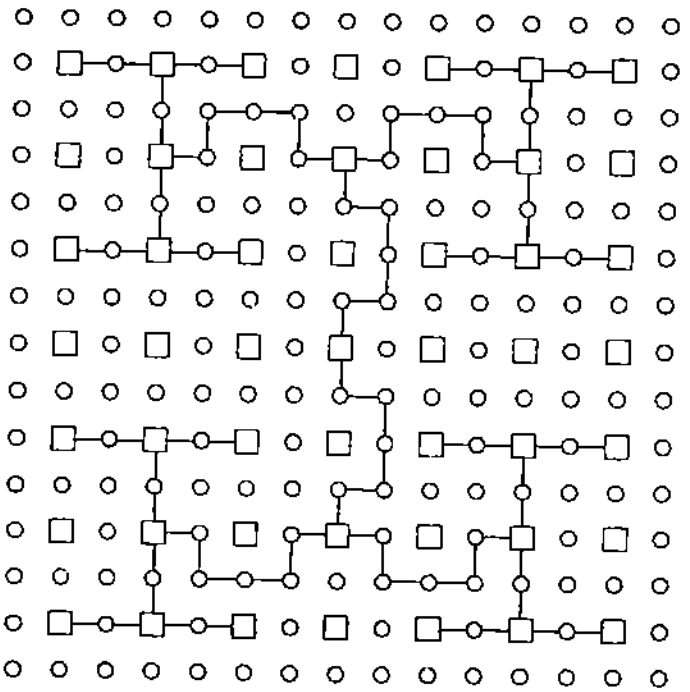


Figure 6. The hyper-H tree (Figure 1(d)) directly embedded into the switch lattice of Figure 2(a); the switches are not shown.

If we adopt the strategy of the hyper-H embedding and locate the root at the center of the tree, then it makes sense to place a spare at the middle of one side so that when this tree is paired to form the next larger tree, there is a spare at the interface ready to become the new root. This will be in the center of the new tree as we intend. (Of course, since the sides always have an even number of PEs, "middle" here means adjacent to the midpoint of one side.) But we cannot pair two trees with their spares in the middle of one side since this will leave us with either a buried spare that is difficult to use when forming the next larger tree or it will leave us with a spare on the perimeter at a site inappropriate for the embedding of the next larger tree. (See Figure 7.)

The solution is to pair one subtree with a spare located at the middle of one side with a subtree whose spare is at the corner. The spare in the middle becomes the root of the new tree and the corner spare

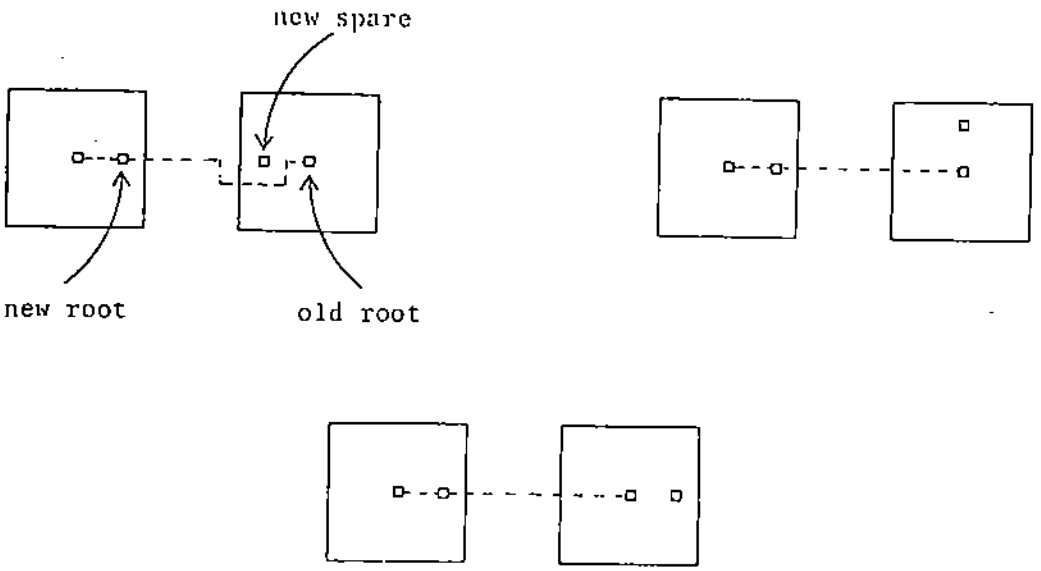


Figure 7. Pairing subtrees using spares located at the midpoint of one side.

can be located (using reflection) to become either a middle spare or a corner spare of the new tree depending on which is needed for the next inductive step. Thus, at each step in the induction we must use (and we can create) two types of embeddings: middles and corners. (See Figure 8.) Notice that the basis tree, embedded in a 2×2 portion of the lattice, actually serves as both types.

Trees, of course, are planar; that is, they can be embedded in the plane without crossovers. But if the reader endeavors to follow the preceding algorithm with the lattice in Figure 2(a), it will appear as though crossovers are required, at least during the early stages of the embedding. It is possible, using basis elements of fifteen node trees

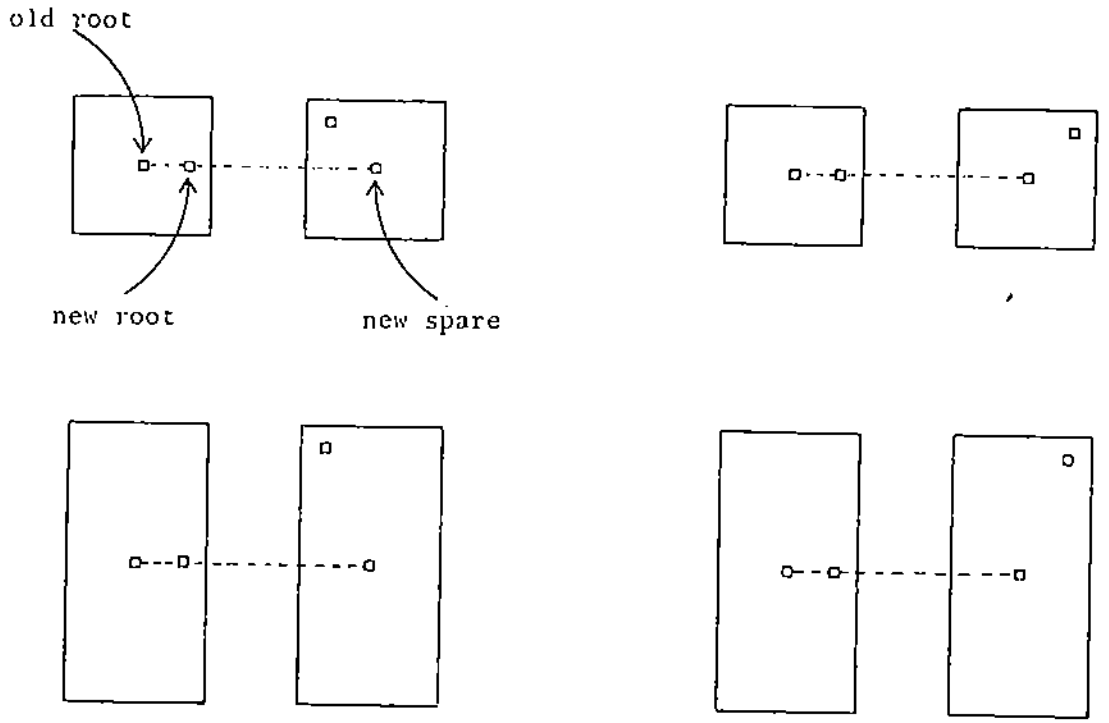


Figure 8. The formation of "middles" and "corners" embeddings using a middle and corner pair.

embedded in 4×4 square regions of the lattice, to achieve a completely planar embedding. A solution is shown in Figure 9 and is completely described in reference [15].

Solving a System of Linear Equations

In order to illustrate how the CHiP processor can be used to compose algorithms, we pose the problem of solving a system of linear equations, i.e. to solve $Ax = b$ for an $n \times n$ coefficient matrix A of bandwidth p and n vector b. We shall use two algorithmically specialized processors

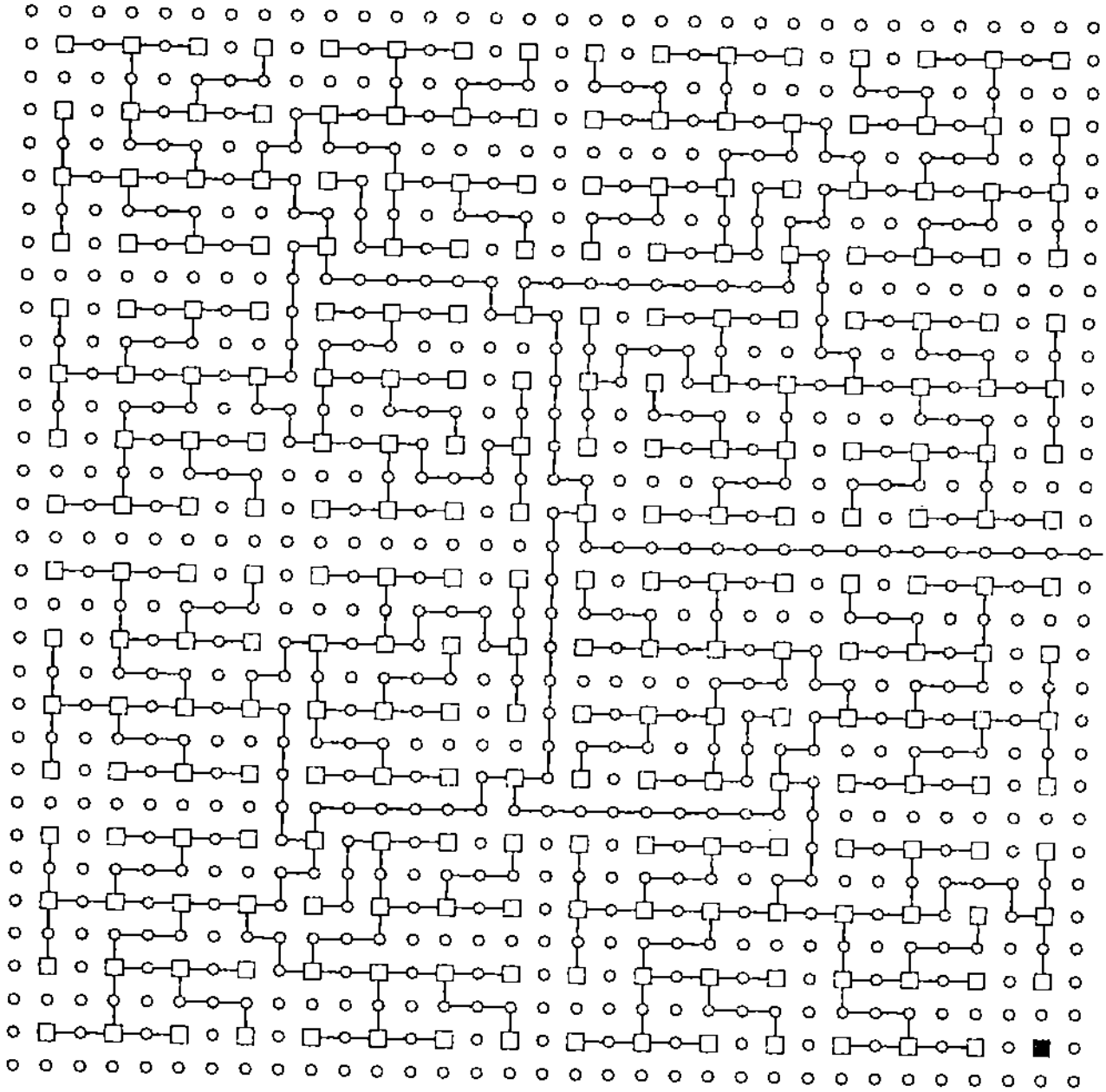


Figure 9. Planar embedding of a 255 node complete binary tree into the lattice of Figure 2(a).

due to H.T. Kung and C.E. Leiserson as described in Mead and Conway [1]. The first is an LU-decomposition systolic array processor that factors A into upper and lower traingular matrices U and L.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & \ddots & \\ & a_{52} & a_{53} & & \\ 0 & & & & \end{bmatrix} = \begin{bmatrix} 1 & & & & 0 \\ \ell_{21} & 1 & & & \\ \ell_{31} & \ell_{32} & 1 & & \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 & \\ & \ell_{52} & \ell_{53} & \ddots & \\ 0 & & & & \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & & 0 \\ & u_{22} & u_{23} & u_{24} & u_{25} & \\ & & u_{33} & u_{34} & u_{35} & \\ & & & & \ddots & \\ 0 & & & & & \end{bmatrix}$$

The second systolic processor solves a lower triangular linear system $Ly = b$ where L is the output from the decomposition step. (We call this the LTS solver.) The final result vector x can be found by solving $Ux = y$ where U is the upper triangular matrix from the first step and y is the vector output of the second step. By rewriting U as a lower triangular system we can use another instance of the LTS solver. Our approach will be to compose these pieces into a harmonious process to solve the entire problem.

The first problem we must solve is the embedding of the Kung-Leiserson systolic processors. These algorithmically specialized processors are defined for $n \times n$ arrays of bandwidth p. (Figure 10 shows the LU-decomposition processor for a $p = 7$ system. Figure 11 shows a suitable lower triangular system solver processor.) Since the LU-decomposition processor is hexagonally connected, it will be convenient to embed the processors into the lattice shown in Figure 2(b). The obvious strategy

is to connect the processors in such a way that the lower triangular output L of the decomposition step connects directly to the input of the lower triangular system solver. It is also obvious that these embeddings should be placed at the perimeter of the CHiP lattice so that matrix A and vector b can be received from external storage. Figure 12 shows such an embedding* where the PE labellings correspond to those given in Figures 10 and 11.

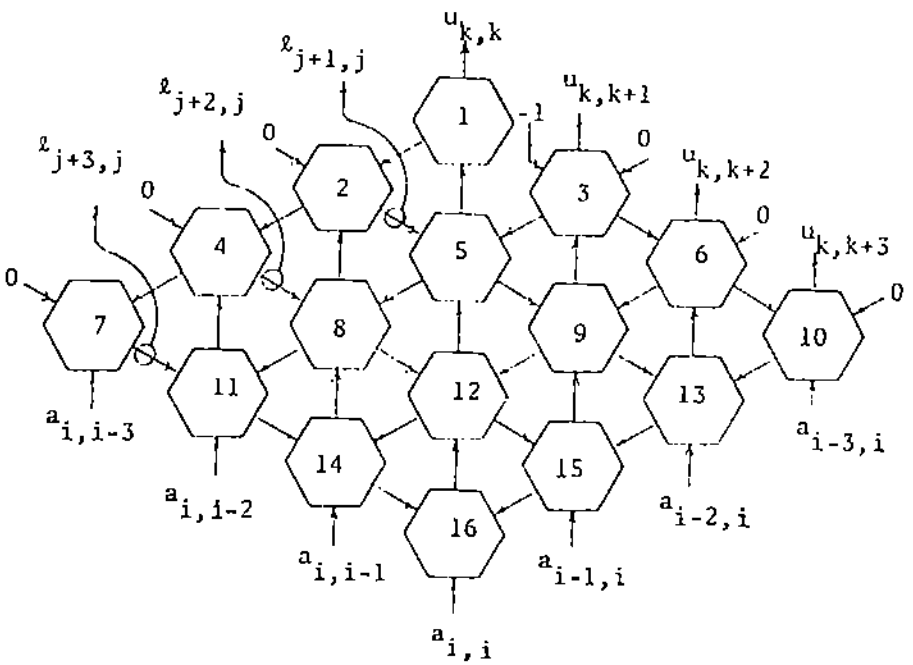


Figure 10. The Kung-Leiserson systolic array for LU-decomposition. Labellings indicate data paths. For timings, see reference [1].

* Although the data paths are bidirectional, we have used arrows to emphasize the direction of data movement.

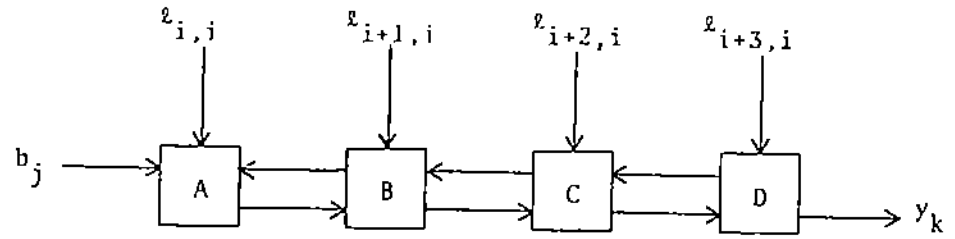


Figure 11. The Kung-Leiserson systolic LTS solver for $w=4$. Labellings indicate data paths for elements of L and b. For timings, see reference [1].

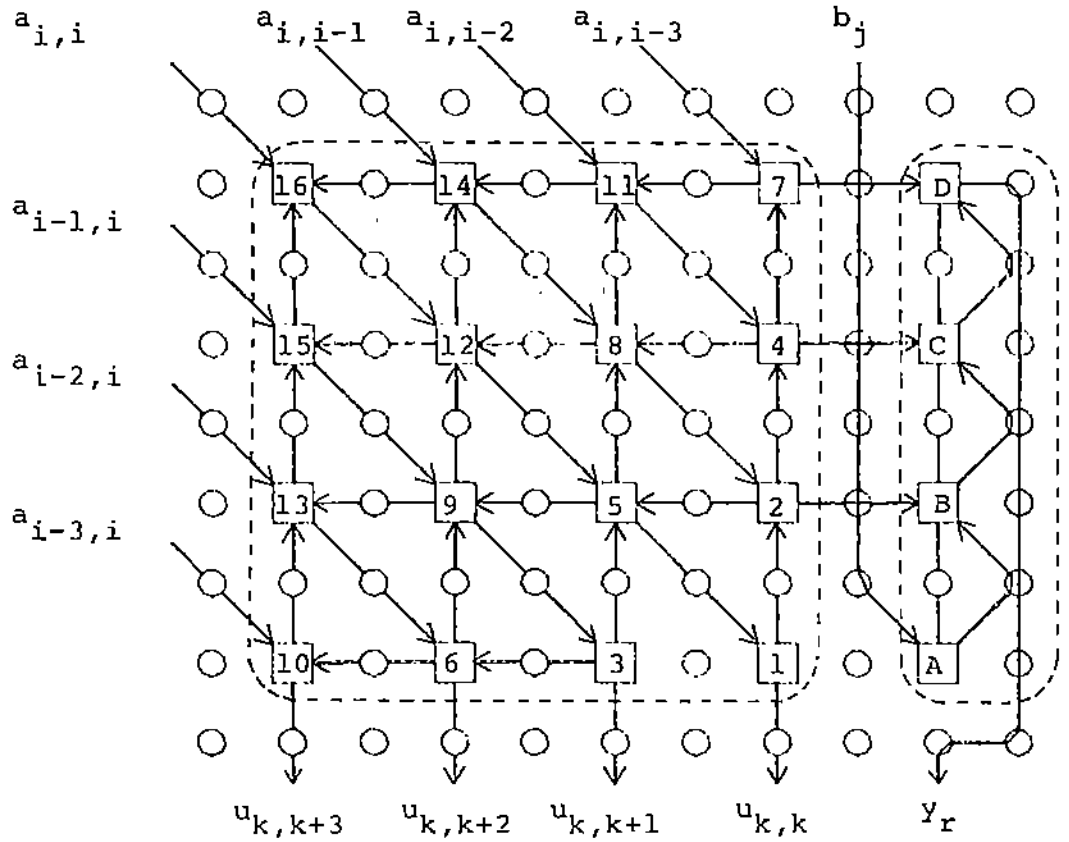


Figure 12. The embedding of the LU-decomposition processor and the LTS solver in the lattice of Figure 2(b). PE labellings correspond to Figure 10 and 11.

Several simple transformations have been employed to accomplish the embedding. The most noticable is that the hexagonal structure has been slightly deformed to accomodate the rectangular CHiP lattice and the LU-decomposition processor has been rotated clockwise 120°. The constant inputs (0's and -1) that appear on the perimeter of the systolic array have been suppressed since they can be generated internally to the PEs. The output wires carrying the L matrix result have been assigned to one of the available ports and routed to the inputs of the LTS solver. Finally, to embed the double channel between PEs of the LTS solver we have routed data diagonally out of the North-East port into the South-East port. Notice that since the diagonal elements of L are all 1, they are not explicitly produced.

The next problem to solve is the rewriting of U as a lower triangular system suitable for input into another embedded LTS solver. We must wait until U has been entirely produced before performing this operation. So, rather than writing the elements of U to external storage as they are produced, we thread them through the lattice (assuming there is sufficient space to store them all). We also thread the y vector output from the LTS process along with U. Then in the second phase of our algorithm, we can process the elements through another embedded LTS solver.

Perhaps the most elegant way to thread U and y through the lattice is to use a graph embedding due to Aleliunas and Rosenberg [13]. The scheme has the advantage of not requiring a large "bundle" of wires along the perimeter of the lattice when the threads double back. (Figure 13 illustrates the embedding required for doubling back.) As the U and y values are produced, they are passed from PE to PE. (They could be

"concentrated" by storing several per PE.) When U and y are completely produced, the first phase is completed.

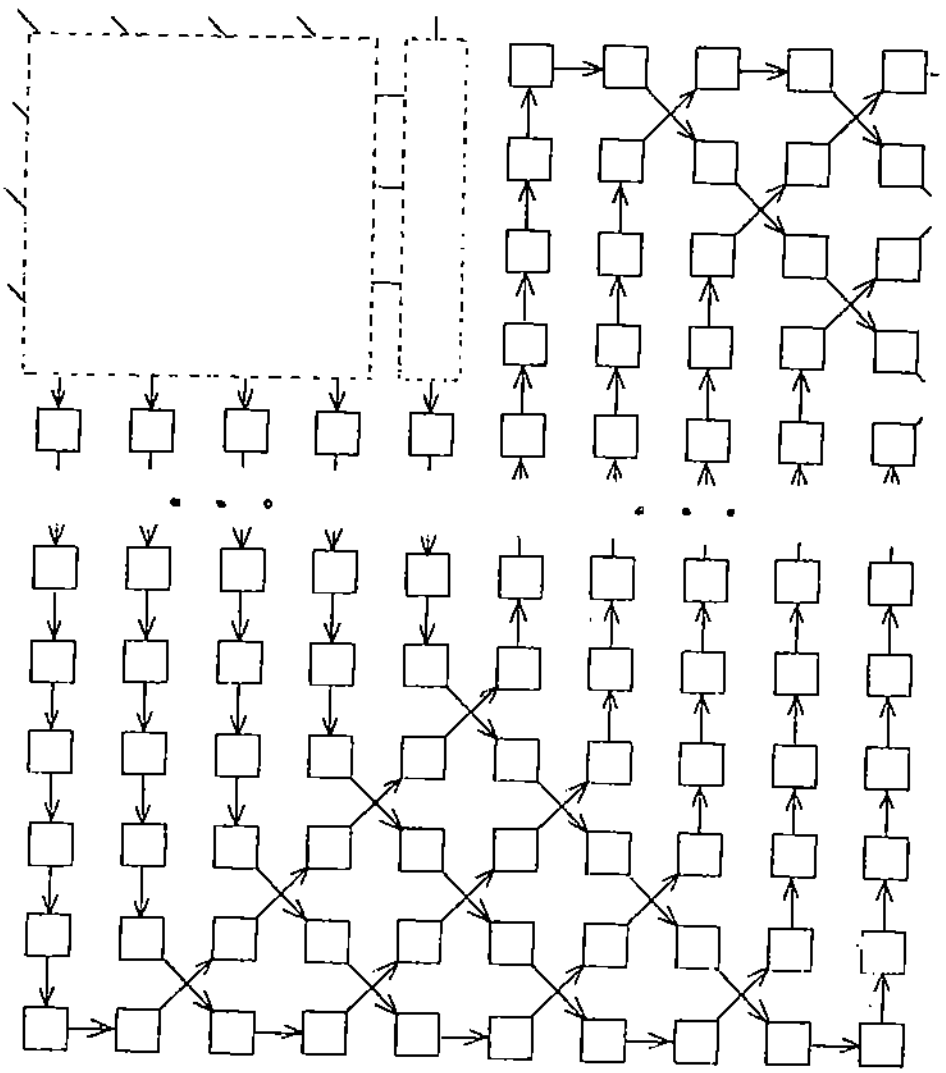


Figure 15. The Abeliunas-Rosenberg embedding of the threads doubling back. The arrows indicate the direction of flow of the U and y values.

Between the first and second phases we make a minor reconfiguration. (This reconfiguration would not have been necessary had the phase 1 configuration been somewhat more clever; but as an example, it would also have been somewhat more confusing.) The second configuration embeds the LTS solver into the fourth row of processors as illustrated in Figure 14.

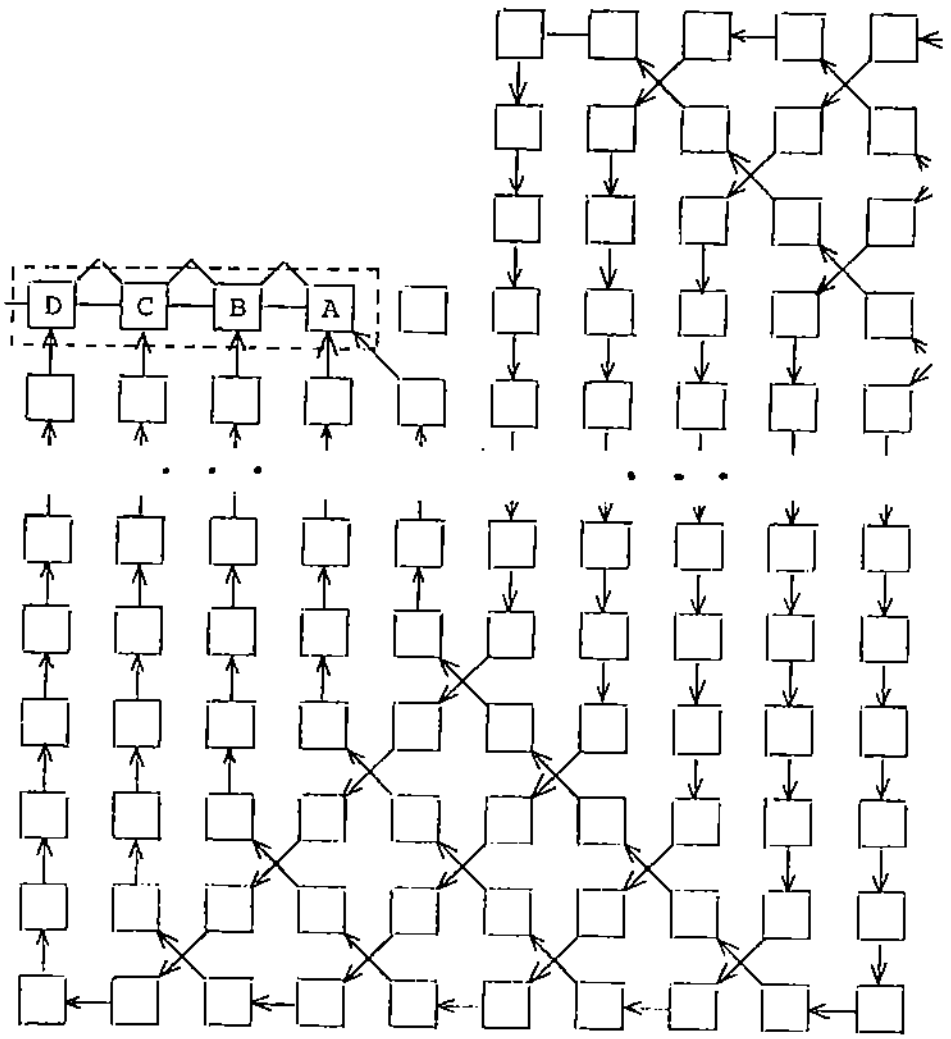


Figure 14. The simple phase 2 embedding

The inputs to this group of processors come from reversing the direction of flow of the threaded values from phase 1. Notice that this reversal of flow has the effect of renumbering the matrix U to be in lower triangular form appropriate for the LTS solver. The appropriate values of the y vector are also available at the proper locations. The outputs from the second phase emanate from the western port of processor (4,1). These are the values solving $Ax = b$.

To summarize, the system of linear equations $Ax = b$ is solved in two phases on the CHiP processor. In phase 1 an embedded LU-decomposition processor takes A as input and produces matrices L and U as output. The L output is immediately input to an LTS solver that also takes b as input and solves $Ly = b$. The vector y and the matrix U are threaded through the lattice. Phase 1 completes when A has been decomposed. In phase 2 another embedded LTS solver takes the threaded output from phase 1 (by reversing its flow) and solves $Ux = y$.

Phase 2 makes scant use of parallelism - it runs in the same time as phase 1 and the data are already in the CHiP processor. And as noted, the interphase reconfiguration was not essential. But, there *are* algorithms to solve the phase 2 problem that do make essential use of configurability to make effective use of parallelism [14]. A complete development of the approach is not possible here, but the essential idea due to Chen, Kuck and Sameh [11] is straightforward: A transformation on U enables us to decompose the matrix into blocks $B_1,\ldots,B_k$ whose product yields the result. Because the product operation is associative, the whole product can be formed by taking pairwise products in parallel, then pairwise products of the results, etc. By reconfiguring the threaded portion of the lattice using one of several rather complicated interconnection patterns that

either implicitly or explicitly embed a tree, we can perform these pairwise products in parallel. The result is a faster parallel algorithm made possible by configurability.

Discussion

Several characteristics of the CHiP approach should be mentioned.

First, the algorithmically specialized processors translate *mutatis mutandis* to programs for the CHiP computer. Thus, we have a ready supply of algorithms that can effectively use the parallel processor. Of course, all of these algorithms use one interconnection structure, and it is possible that improved algorithms might be found that exploit the availability of multiple interconnection structures.

Second, configurability provides both interphase and intraphase flexibility. This distinction, though not very clear-cut, tends to correlate with whether or not pipelining is being used. If a problem is solved by a sequence of phases that each complete before the next one begins, we tend to use regular configurations that change at the completion of a phase (interphase). The whole lattice is in a mesh or tree pattern. For a series of pipelined algorithms that can be coupled together, as in the last section , we tend to form regions of the lattice dedicated to each algorithm with data paths interconnecting the regions. We refer to this as intraphase configurability because within one phase we interconnect several regular structures. Clearly, we need not *change* configurations to exploit the advantage of configurability.

Both kinds of configurability are useful in adapting to changes in problem size. For example, two different small problems might operate concurrently on different regions of the CHiP processor using entirely different interconnection schemes. One pattern could change while the

other remained fixed by loading switches of the fixed region with two copies of the same configuration setting. Pipelined processors, whose size is usually a function of the input width, can be tailored to the right size at loading time.

Another consequence of configurability is that it is quite fault tolerant. Supposing that an error is detected in a processor, data path or switch, we can simply route around the offending device. For convenience, we might choose to leave other processors unused to "square up" the lattice when matching dimensions are important.

Perhaps the most intriguing consequence of configurability's fault tolerance is the possibility of "wafer level" fabrication. That is, instead of dicing a wafer and discarding the faulty processor chips, we can leave a VLSI wafer whole and simply route around the unusable processors. (We could use the dicing corridors for data paths, and switches.) For example if a wafer contains 100 processor chips and yield characteristics indicate that roughly one third are faulty, then a wafer is acceptable if we can find an 8×8 sublattice that is functional. The mapping of the switches to host the 8×8 in the 100 could be done on the wafer by special circuitry designed for that purpose. Although the number of pins required for the wafer would be large, their number is only proportional to the perimeter rather than the area. This actually reduces the total number of wires bonded.

Summary

By integrating programmable switches with the processing elements, the CLiP computer achieves a polymorphism of interconnection structure that also preserves locality. This enables us to compose algorithms that

exploit different interconnection patterns. In addition to responding to different problem sizes and characteristics, the flexibility of integrated switches provides substantial fault tolerance and permits wafer level fabrication.

Acknowledgements

It is a great pleasure to thank Dennis Gannon for his encouragement and his assistance with the linear systems solving example. Janice Cuny's critical reading has lead to a simplification of the switch - the insight is much appreciated. Thanks are due Paul McNabb who developed programs to produce the embedding of Figure 9. Finally, Robert Grafton, Leonard Haynes and Richard Lau have provided encouragement and support that is greatly appreciated.

References

[1] Carver Mead and Lynn Conway
Introduction to VLSI systems
Addison Wesley, 1980

[2] H.T. Kung and C.E. Leiserson
Systolic arrays (for VLSI)
Tech. Report CS-79-103, Carnegie-Mellon University, April 1979
(Also in [1])

[3] D.B. Gannon
On pipelining a mesh connected multiprocessor for finite element problems by nested dissection
Proc. Int'l Conf. on Parallel Processing, pp. 197-204, 1980

[4] Sally Browning
The tree machine: a highly concurrent programming environment
Ph.D. Thesis, California Institute of Technology, Jan. 1980

[5] Jon L. Bentley and H.T. Kung
A tree machine for searching problems
In Proc. of the Int'l Conf. on Parallel Processing, pp. 257-266
IEEE, 1979

[6] L. Snyder
Tree-organized processor structure
Technical Report, Yale University, March 1980

[7] L.J. Guibas, H.T. Kung and C.D. Thompson
Direct VLSI implementation of combinatorial algorithms
In Cal. Tech. Conf. on VLSI, California Institute of Technology
January 1979

[8] S.W. Song
A highly concurrent tree machine for data base applications
Proc. Int'l Conf. on Parallel Processing pp. 259-268, 1980

[9] L.G. Valiant
University considerations in VLSI circuits
IEEE Trans. Computers, 1981

[10] C.D. Thompson
A complexity theory for VLSI
Ph.D. Thesis, Carnegie-Mellon University, 1980

[11] S.C. Chen, D.J. Kuck and H.H. Sameh
Practical Parallel Based Triangular System Solvers
ACM TOMS (Sept. 78) pp. 270-277.

[12] L. Snyder
Overview of the CliP Computer
In VLSI 81, John Grey, ed., Academic Press, pp. 240-249, 1981

[13] Romas Aleliunas and A.L. Rosenberg
On embedding rectangular grids into square grids
IBM Tech. Report RC 8404 1980

[14] D.B. Gannon and L. Snyder
Linear Recurrence Algorithms for VLSI: The
Configurable, Highly Parallel Approach
(in preparation)

[15] Lawrence Snyder
Programming Processor Interconnection Structures
Purdue Universities Department of Computer Sciences, TR-381, 1981